Time Series Database Performance Comparison Using GridDB and InfluxDB

March 12, 2018
Revision 1.9

Table of Contents

Table of Contents 1
List of Tables and Figures 1
Executive Summary 2
Introduction 2
Environment 3
AWS Configuration 3
Software 4
Configuration 4
GridDB 4
InfluxDB 5
Test Methodology 5
Test Design 5
Methodology 6
Collection and Aggregation 6
Results 7
Insert 7
Read 8
Scan 9
Database Size 11
Tabular Results 11
Conclusion 13
Appendices 14
gs_node.json 14
gs_cluster.json 15
influxdb.conf 15

List of Tables and Figures

Figure 1: Insert Throughput 8
Figure 2: Read Throughput 9
Figure 3: Total Scan/Aggregation Throughput 10
Figure 4: Scan, Count, Average, and Sum Latencies. 10
Figure 5: Storage Requirements 11
Table 1: AWS Instance Specifications 4
Table 2: YCSB-TS Database Schema 6
Table 3: Workload Throughputs 11
Table 4: Operation Latencies 12
Table 5: Storage Requirements 12
Table 6: CPU/Memory Usage. 13

Executive Summary

With the Internet of Things (IoT) projected to have 50 billion plus operating devices generating one trillion dollars in revenue and even more Time Series data, Time Series database (TSDB) performance is becoming crucial for organizations developing applications to leverage this data. Fixstars decided to evaluate two open source, innovative Time-Series Database (TSDB) products that have been gaining mindshare in the market. These databases were Toshiba Digital Solutions Corporation's GridDB and InfluxData's **InfluxDB –** both touted for their IoT focused architecture.

Both databases were tested with Yahoo Cloud Servicing Benchmark – Time Series (YCSB-TS) running on Amazon Web Services (AWS). We compared ingestion, read, and scan performance to cover the generic operations of time series databases. In addition, we also highlighted other important criteria like latency and queries – which are the other key parameters in IoT use cases. One workload was used to test the read performance with read queries to search for timestamps. Other workload covered scan, count, average, sum of timeseries data. Two different workloads were tested with datasets of 100 and 400 million records.

The results of the tests surprised us by showing that GridDB markedly outperformed InfluxDB in both latency and throughput. InfluxDB has been around a few years and has gained a nice niche market-share in IoT use cases. Yet, GridDB's innovative architecture of intelligently using containers (containers are similar to the tables of Relational Database) bears results. The results also demonstrated GridDB's superior scalability and consistency, which are both thanks in part to GridDB's in-memory architecture and ACID (Atomicity, Consistency, Isolation & Durability)) compliance within the container.

Introduction

In data management, time series data can be defined as a sequence of values collected over a time interval. Examples of time series data are IoT sensor data, health monitor information, solar flare tracking, event monitoring, etc. Most traditional NoSQL databases fail to provide the performance and scalability needed to handle large volumes of timeseries data. As a result, databases that specialize in handling timeseries data were developed. **TSDBs such as GridDB or InfluxDB** are databases specialized for storing, collecting, retrieving, and processing timestamped data. Timeseries databases are optimized to provide effective data compaction, high-write performance, and fast range queries. This makes timeseries databases more scalable, reliable, and cheaper than traditional databases for processing timestamp data.

GridDB is an in-memory oriented, distributed NoSQL database with a hybrid cluster architecture to provide reliability and is one of the only marquee NoSQL databases to also be fully ACID compliant. GridDB can be used as a time series database by housing data in timeseries containers. These timeseries containers provide time-type indexing and data functions like time-weighted averaging and interpolation. Timeseries containers also provide a unique compression utility to effectively

release expired data. GridDB is available under both open source and commercial licenses. The version used for these tests was the open-source variety, GridDB Community Edition.

InfluxDB is a Time Series database and is built to handle high write and query loads. It uses a HTTP Representational State Transfer (REST) API for querying data as well as an SQL-like query language known as *InfluxQL*. InfluxDB possesses a distributed architecture in which multiple nodes can handle storage and execute queries simultaneously. InfluxDB uses a Time Structured Merge (TSM) Tree storage engine to handle high ingest speed and perform effective data compression. InfluxDB has open-source editions and commercially licensed enterprise editions available for use. For benchmark testing, InfluxDB Open Source Edition was used.

YCSB-TS is a software-fork of the modular benchmark YCSB, Yahoo! Cloud Serving Benchmark that is optimized for testing timeseries NoSQL databases. Like the original YCSB, it is also written in Java. YCSB-TS supports timeseries ranges, makes use of time-domain functions, and adds workload options that are specific to timeseries databases.

Environment

AWS Configuration

The YCSB-TS benchmark tests were run on a C4 AWS EC2 instance based on a CentOS 6.9 image. C4.2xlarge model was used as it is optimized for compute-intensive workloads.

The AWS instance held both database servers although only one was tested at a time. A database server ran with the YCSB-TS client. The testing process consisted of starting and connecting to the database server, running a YCSB-client, collecting and aggregating performance data along with resource usage.

For both InfluxDB and GridDB, all their data was stored on a 50GB io1 Elastic Block Storage with 1000 IOPS provisioned that ensured more consistent results over a general-purpose (gp) block storage type device with burstable performance.

AWS Instance Type	C4.2xlarge
Operating System	CentOS 6.9
CPU Processor	Intel Xeon CPU E5-2666 v3
vCPU Cores	8
Clock Speed	2.9GHz
Main Memory Size	15GB
Data Storage	100GB io1 EBS with 1000 IOPS provisioned

Table 1: AWS Instance Specifications

Software

GridDB Community Edition version 3.0.1 was installed into the AWS instance using RPM package obtained from the official GridDB website (www.griddb.net). InfluxDB version 1.3.6 was also installed using the official RPM packages downloaded from InfluxData's website (www.influxdata.com).

YCSB-TS was cloned from the official TSDBBench GitHub repository on October 2017. The full YCSB-TS distribution along with all its bindings were built using Maven. The InfluxDB driver for YCSB-TS was modified from its original source so that it could distribute datasets between multiple measurements. A custom database connector for GridDB was developed using the GridDB driver for the original YCSB framework as its base.

Configuration

GridDB

In the configuration file, gs_node.json, "storeMemoryLimit" was increased 6192MB (6GB) to allow for the small data set to fit within storeMemory. Concurrency was increased to 8 to match the number of vCPUs.

The field for storeCompressionMode was also added and set to BLOCK_COMPRESSION. This allowed GridDB to minimize storage use when storing large datasets. Block compression in GridDB involves exporting in-memory data to be compressed, allowing vacant areas of memory to be deallocated, thereby reducing disk use.

InfluxDB

Issues that can arise with InfluxDB include high memory usage and lower performance when using a high cardinality dataset. To prevent this, adjustments were made to InfluxDB's configuration file, influxdb.conf. For example, http-logging was disabled so that logs of all POST and insert operations would not be stored, reducing disk usage.

Max-select-point and max-select-series were disabled in influxdb.conf to insure that read and scan queries would not fail against timeseries that stored millions of data points. . Other changes included setting cache-snapshot-write-cold-duration to 10 seconds and setting the index-version used to **tsi-1** to improve performance with larger dataset.

InfluxDB's TSM storage engine uses compression by default.

Test Methodology

Test Design

It was determined that a thread count of 128 threads would be used to execute the benchmark tests for load and run phases; 128 threads usually provide the most consistent performance. It was determined that two datasets of different sizes would be used to test each database's scalability and performance when operating on a high cardinality container. A large dataset is useful in observing how each database compresses their data. One dataset of 100 million records was used in the first phase of testing and a larger dataset of 400 million records was used in the following phase. The records would be spread amongst eight containers or metrics to match the number of vCPUs in the instance. Each container or metric would contain 12.5 or 50 million records respectively.

The 100 million record dataset was roughly 5 GB in size and would remain in memory. The 400 million record dataset was about 20GB in size and approximately 60% would reside on disk with GridDB. With InfluxDB, the kernel's disk cache management system would control what data was flushed to disk or kept in memory.

Every record inserted by YCSB-TS would consist of a timestamp as the row key, three string tags, and a double value that represents a metric reading. Each string tag would be 10 characters long, totaling 10 bytes each. This would make each record roughly 50 bytes long. All data fields would be generated at random by YCSB-TS. The range for timestamps in large datasets would span roughly from a time range of 1-4 days with at least millisecond between each timestamp record.

Record Calculation:

$$(\,12\;byte\;timestamp) + (3 * (10\;byte\;string\;tag\,)) + (8\;byte\;double\;value) = 50\;bytes$$

Column Name	Column Type	Data Size	Example
Time	Timestamp (Row-key)	12 bytes	1439241005000
TAG0	String	10 bytes	“Wh64HSAlAU”
TAG1	String	10 bytes	“dXannLxHhW”
TAG2	String	10 bytes	“wTRxj0tNW9”
value	Double	8 bytes	5423.22

Table 2: YCSB-TS Database Schema

Methodology

To ensure consistency, three trials were run on the AWS instance to measure the throughputs and latencies for each dataset. The median throughput and latency were taken from the three trials. The reason for this decision was to ensure that all measurements were due to performance differences between InfluxDB and GridDB, not from AWS.

The head node would begin in a “fresh, deallocated” state. Once the AWS instance began running, the local SSD and database directories would be mounted. The AWS instance began by wiping all the containers and its data and logs from each database and by deploying their configuration files. From there, either InfluxDB or GridDB would be started through an init script or a database command.

When running a test against GridDB server, settings and statistics would be recorded with the gs_stat command; the Influx shell was used for InfluxDB.

The YCSB-TS load operation is executed on the client node with the appropriate insertstart, insertend, and recordcount parameter values. These values are adjusted depending on the workload configuration and dataset size. After the load phase completes, one of the two workloads are run:

Workload A: Read only
Workload B: Scan, Count, Average, and Sum operations.

More information on the YCSB-TS architecture and usage can be found on the YCSB-TS GitHub page:
https://github.com/TSDBBench/YCSB-TS

Collection and Aggregation

Statistics related to the database size, CPU usage, and memory usage are captured using Bash scripts after the workloads are finished.

All the output from YCSB-TS was captured for processing with bash scripts that are outputted into log files. The data in these log files were used for later processing in spreadsheet programs. Important output data to monitor both during and after YCSB-TS operation are database disk usage and how its data is distributed among its *data, log, or wal* directories. The purpose of monitoring this data was to observe database size and compression.

Output from YCSB-TS was captured for later processing using bash scripts that are redirected to log files. The data from these logs were used for later processing in spreadsheet programs. Data metrics like CPU usage, memory usage, and disk usage were recorded using bash scripts that redirected output from top and other commands to log files as well.

Results

Insert

The timestamp range for the benchmarks would range from 1 to 4 days. Each record would have a minimum of 1 millisecond between insertions. Dataset size would range from 100 million to 400 million records. The smaller dataset of 100 million records was used to test how each database performs with the dataset in-memory. For GridDB, due to its in-memory architecture, all of the 100 million records fit in-memory rather than having to be read from or written to disk cache. On the other hand, the larger dataset was used to test how each database performs with a large portion of its data residing on-disk. A dataset of 400 million rows would have over two-thirds of its data stored and accessed on-disk. Both databases were configured to use compression to minimize use of secondary storage. To prevent Java from running out of heap space, the 'predefinetagstoreused' field was set to false in all workload configuration files.

Run phases of the benchmark tests would be executed an hour after all YCSB-TS records were inserted into either database. The purpose of this delay was to let each database complete all housekeeping operations to ensure fairness during read and scan benchmarks.

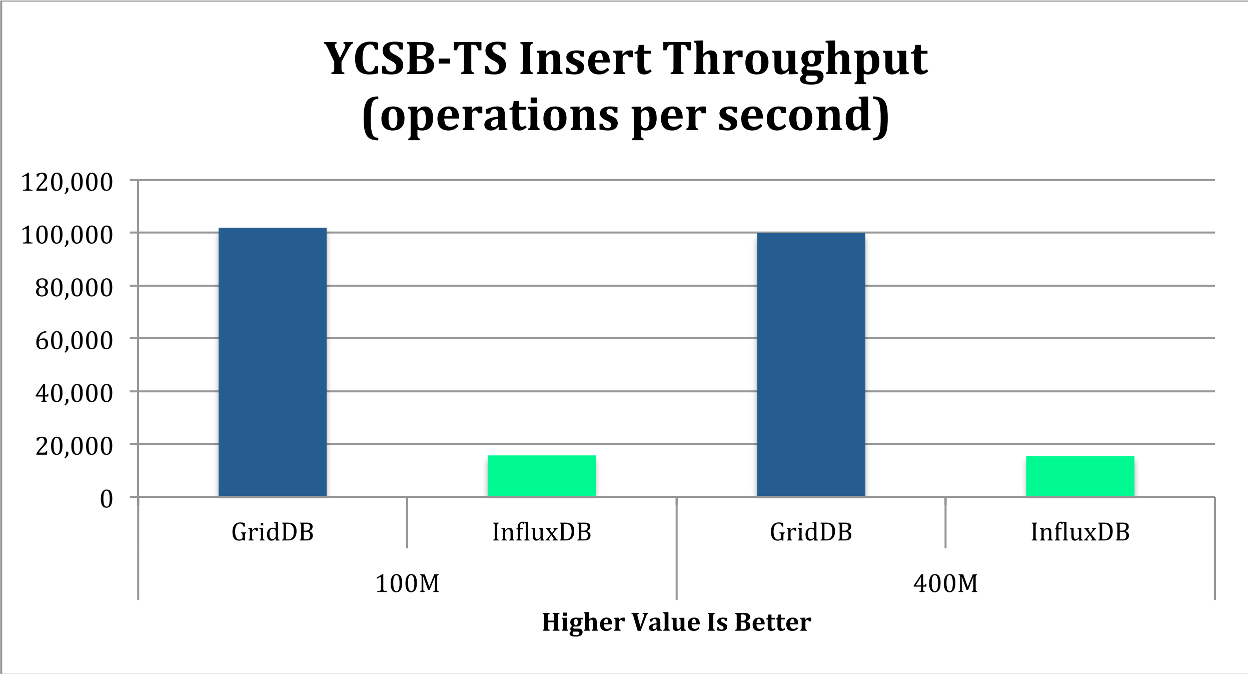

Figure 1: Insert Throughput

Read

Workload A was a workload configuration that consisted only of read operations after all records were loaded in the database. Each read operation would search for a record that had a specific timestamp as its row key. The timestamp that would be searched would be generated at random by YCSB-TS.

For the 100M read test, GridDB was able to perform nearly 8x more operations per second than InfluxDB and for the 400M read benchmark. With the larger 400M record set, GridDB's performance fell by about a factor of four while InfluxDB's read throughput fell by a factor of five giving GridDB a near 10x performance advantage.

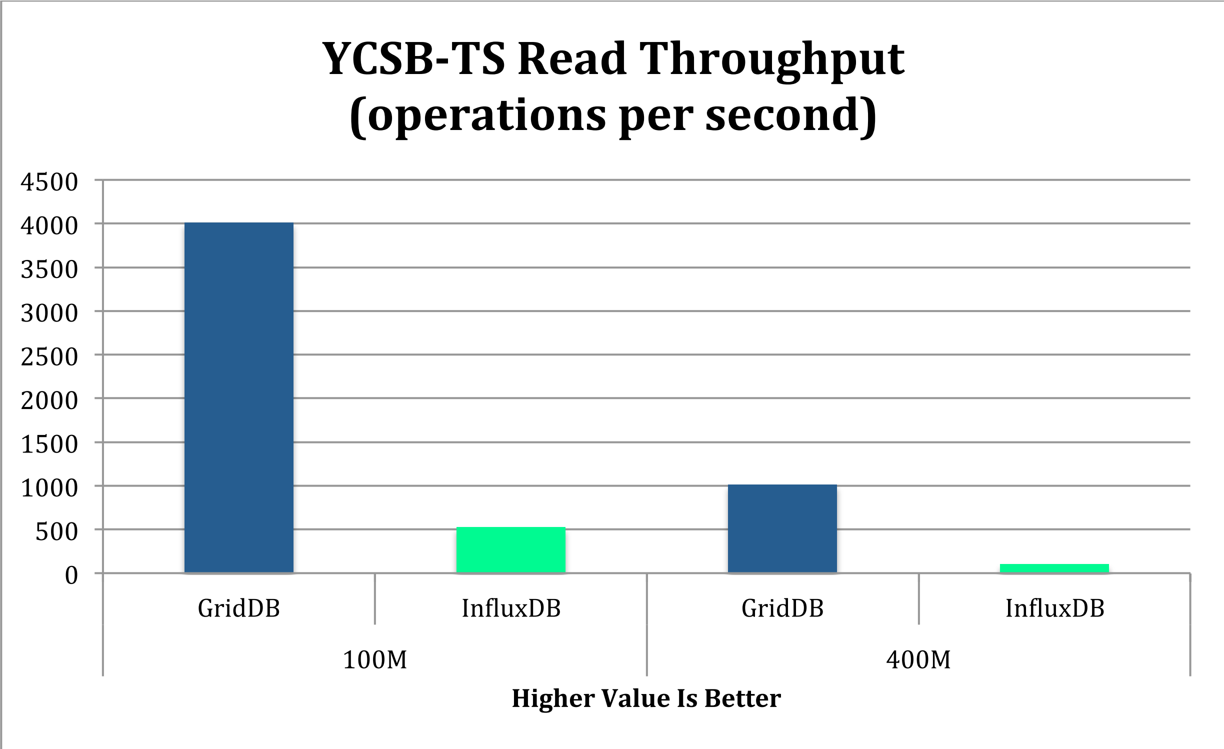

Figure 2: Read Throughput.

Scan

Workload B was a workload configuration that consisted 100% of scan operations. 25% of these operations were general Scan searches, 25% were Count operations, 25% were Sum operations, and 25% were Average operations.

The general SCAN operation would search for rows between two randomly generated timestamp values. The COUNT operation would count how many rows in a timeseries container are between two timestamp values. AVG and SUM would calculate the average or sum of the double values of every row found between the time range respectively. These timestamp values used as the query range would always be between insertstart and insertend fields specified in the workload configuration files.

YCSB-TS only reports the throughput of scan operations during the run phase. However, it does report the latencies of the Average, Sum, Count, and Scan operations.

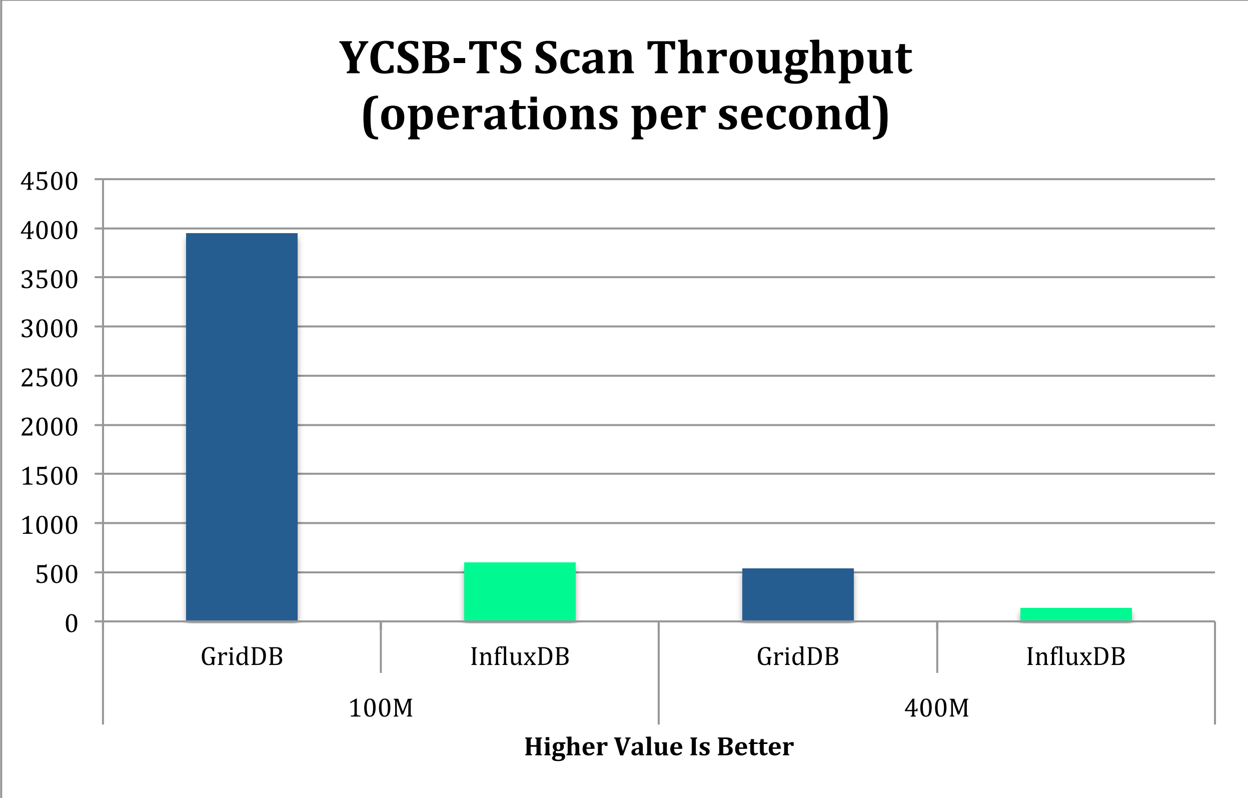

Figure 3: Total Scan/Aggregation Throughput.

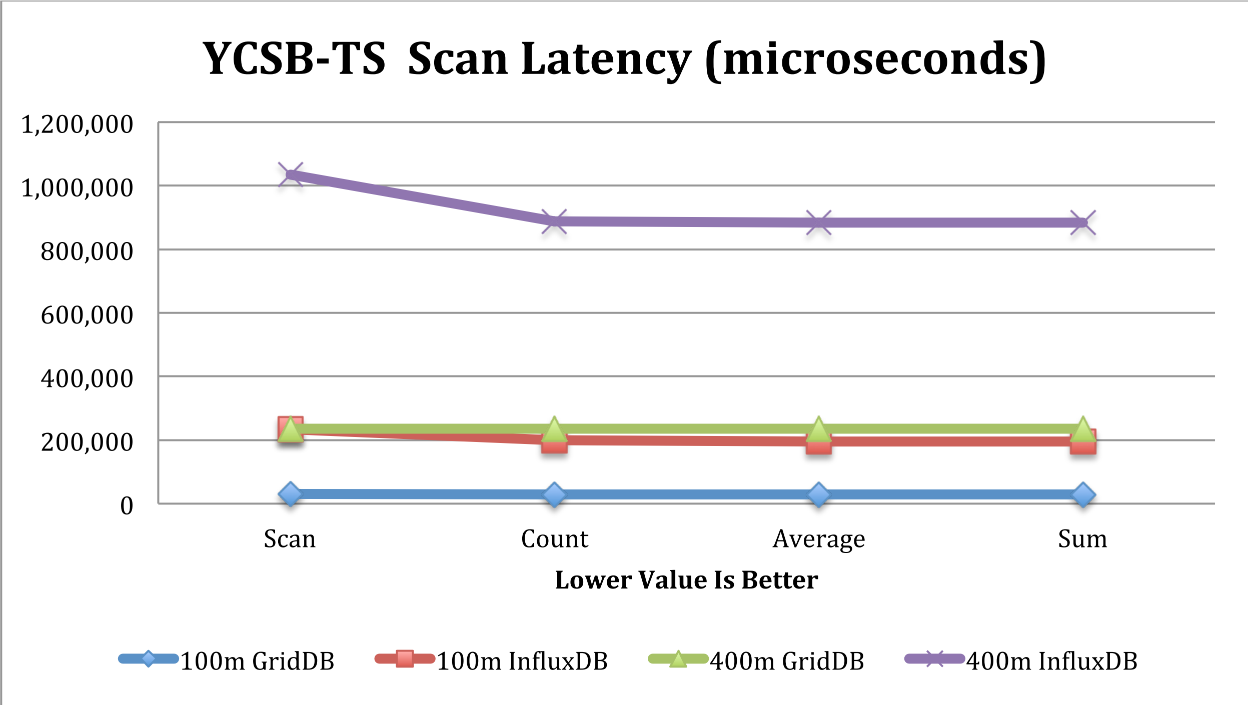

Figure 4: Scan, Count, Average, and Sum Latencies.

Database Size

To compare the size efficiency of GridDB and InfluxDB's their size of the on-disk data was measured immediately after a load was completed and after the Write-Ahead-Log Files are flushed.

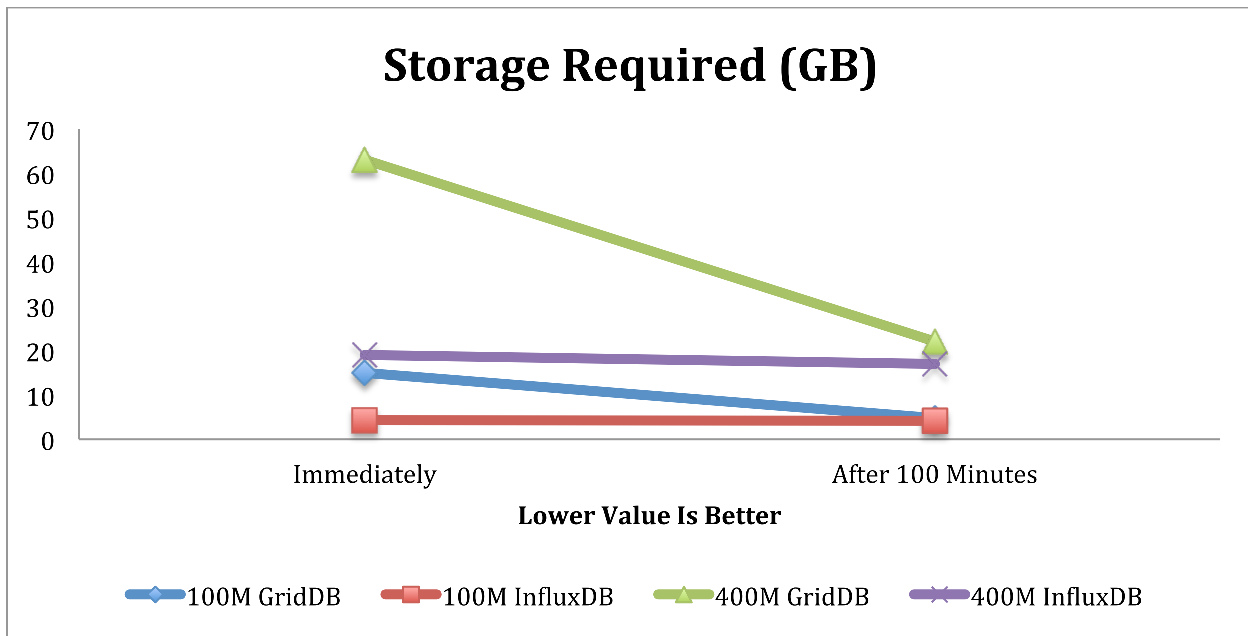

Figure 5: Storage Requirements

Tabular Results

Charts and Spreadsheets

All throughput measurements are in the units of "operations per second" and all latency measurements are in the units "microseconds".

Throughput (operations per second)

Test	Size	GridDB	InfluxDB	Advantage
Load	100M	101,793.7	15,637.3	GridDB 651% Better
	400M	99,771.1	15,512.0	GridDB 643% Better
Workload A	100M	4,013.3	525.6	GridDB 764% Better
	400M	1,014.6	102.4	GridDB 991% Better
Workload B	100M	3,948.0	599.6	GridDB 658% Better
	400M	535.8	136.3	GridDB 393% Better

Table 3: Workload Throughputs

Latencies (microseconds).

Operation	Size	GridDB	InfluxDB	Advantage
Load	100M	1,244.4	8,178.7	GridDB 657% better
	400M	1,250.1	8,246.3	GridDB 660% better
Read	100M	28,681.6	236,671.5	GridDB 825% better
	400M	122,946.4	1,141,034.2	GridDB 928% better
Scan	100M	29,711.5	233,252.0	GridDB 785% better
	400M	235,213.9	1,035,776.4	GridDB 440% better
Count	100M	29,130.8	199,063.9	GridDB 683% better
	400M	234,780.9	888,515.0	GridDB 378% better
Average	100M	29,082.4	195,110.7	GridDB 671% better
	400M	234,858.1	882,958.3	GridDB 376% better
Sum	100M	29,068.2	194,563.9	GridDB 669% better
	400M	234,796.2	884,338.7	GridDB 377% better

Table 4: Operation Latencies

Data Storage Size

		0 minutes	100 minutes
100M	GridDB	15GB	4.8GB
	InfluxDB	4.3GB	4.2GB
400M	GridDB	62GB	21GB
	InfluxDB	19GB	17GB

Table 5: Storage Requirements

CPU and Memory Usage

Test	Workload Size	Measurement	GridDB	InfluxDB
Load	100M	CPU Usage	317%	400%
		Memory Usage	5.5 GB	2.7GB
	400M	CPU Usage	305%	405%
		Memory Usage	5.8 GB	10.2GB
Workload A	100M	CPU Usage	288%	735%
		Memory Usage	5.5GB	4.1GB
	400M	CPU Usage	49.6%	722.5%
		Memory Usage	5.6GB	10.4GB
Workload B	100M	CPU Usage	231%	743%
		Memory Usage	5.5GB	5.2GB
	400M	CPU Usage	61.4%	713%
		Memory Usage	5.8GB	10.3GB

Table 6: CPU/Memory Usage.

Conclusion

The in-memory oriented, hybrid-storage architecture of GridDB provided superior performance compared to InfluxDB. GridDB maintained strong performance with in-memory and out-of-memory database operations and also managed to maintain a higher throughput and lower latency than InfluxDB even as its database size grew. GridDB also maintains consistent and reliable performance as its individual timeseries containers grow, meaning it is not necessary to distribute a large dataset among many containers. This makes GridDB more capable of storing a longer and larger range of timestamps.

These benchmark tests demonstrate GridDB's superior ability to adapt to growing datasets and many different hardware configurations. GridDB's high throughput and lower latency show it is a more scalable and flexible timeseries Database.

Appendices

gs_node.json


```
{
	"dataStore":{
		"dbPath":"data",
		"storeMemoryLimit":"6192MB",
		"storeWarmStart":true,
		"storeCompressionMode": "COMPRESSION",
		"concurrency":8,
		"logWriteMode":1,
		"persistencyMode":"NORMAL",
		"affinityGroupSize":4
	},
	"checkpoint":{
		"checkpointInterval":"1200s",
		"checkpointMemoryLimit":"1024MB",
		"useParallelMode":false
	},
	"cluster":{
		"servicePort":10010
	},
	"sync":{
		"servicePort":10020
	},
	"system":{
		"servicePort":10040,
		"eventLogPath":"log"
	},
	"transaction":{
		"servicePort":10001,
		"connectionLimit":5000
	},
	"trace":{
		"default":"LEVEL_ERROR",
		"dataStore":"LEVEL_ERROR",
		"collection":"LEVEL_ERROR",
		"timeSeries":"LEVEL_ERROR",
		"chunkManager":"LEVEL_ERROR",
		"objectManager":"LEVEL_ERROR",
		"checkpointFile":"LEVEL_ERROR",
		"checkpointService":"LEVEL_INFO",
		"logManager":"LEVEL_WARNING",
		"clusterService":"LEVEL_ERROR",
		"syncService":"LEVEL_ERROR",
		"systemService":"LEVEL_INFO",
		"transactionManager":"LEVEL_ERROR",
		"transactionService":"LEVEL_ERROR",
		"transactionTimeout":"LEVEL_WARNING",
		"triggerService":"LEVEL_ERROR",
		"sessionTimeout":"LEVEL_WARNING",
		"replicationTimeout":"LEVEL_WARNING",
```

```
            "recoveryManager":"LEVEL_INFO",
            "eventEngine":"LEVEL_WARNING",
            "clusterOperation":"LEVEL_INFO",
            "ioMonitor":"LEVEL_WARNING"
        }
}
```

gs_cluster.json

```
{
        "dataStore":{
            "partitionNum":128,
            "storeBlockSize":"64KB"
        },
        "cluster":{
            "clusterName":"defaultCluster",
            "replicationNum":2,
            "notificationAddress":"239.0.0.1",
            "notificationPort":20000,
            "notificationInterval":"5s",
            "heartbeatInterval":"5s",
            "loadbalanceCheckInterval":"180s"
        },
        "sync":{
            "timeoutInterval":"30s"
        },
        "transaction":{
            "notificationAddress":"239.0.0.1",
            "notificationPort":31999,
            "notificationInterval":"5s",
            "replicationMode":0,
            "replicationTimeoutInterval":"10s"
        }
}
```

influxdb.conf

```
[meta]
  # Where the metadata/raft database is stored
  dir = "/var/lib/influxdb/meta"

  # Automatically create a default retention policy when creating a database.
  # retention-autocreate = true

  # If log messages are printed for the meta service
  # logging-enabled = true
[data]
  # The directory where the TSM storage engine stores TSM files.
  dir = "/var/lib/influxdb/data"

  # The directory where the TSM storage engine stores WAL files.
  wal-dir = "/var/lib/influxdb/wal"
  # wal-fsync-delay = "0s"
  index-version = "tsi1"
  trace-logging-enabled=true
```

```
  # CacheMaxMemorySize is the maximum size a shard's cache can
  # reach before it starts rejecting writes.
  cache-max-memory-size = 1048576000

  # CacheSnapshotMemorySize is the size at which the engine will
  # snapshot the cache and write it to a TSM file, freeing up memory
  cache-snapshot-memory-size = 26214400

  cache-snapshot-write-cold-duration = "10s"
# compact-full-write-cold-duration = "4h"
  # max-concurrent-compactions = 0
  # The maximum series allowed per database before writes are dropped.  This limit can prevent
  # high cardinality issues at the database level.  This limit can be disabled by setting it to
  # 0.
  max-series-per-database = 0

  # The maximum number of tag values per tag that are allowed before writes are dropped.  This limit
  # can prevent high cardinality tag values from being written to a measurement.  This limit can be
  # disabled by setting it to 0.
  max-values-per-tag = 0

[coordinator]
  # The default time a write request will wait until a "timeout" error is returned to the caller.
  # write-timeout = "10s"

  # The maximum number of concurrent queries allowed to be executing at one time.  If a query is
  # executed and exceeds this limit, an error is returned to the caller.  This limit can be disabled
  # by setting it to 0.
  max-concurrent-queries = 128

  # The maximum time a query will is allowed to execute before being killed by the system.  This limit
  # can help prevent run away queries.  Setting the value to 0 disables the limit.
  # query-timeout = "0s"

  # The time threshold when a query will be logged as a slow query.  This limit can be set to help
  # discover slow or resource intensive queries.  Setting the value to 0 disables the slow query logging.
  # log-queries-after = "0s"

  # The maximum number of points a SELECT can process.  A value of 0 will make
  # the maximum point count unlimited.  This will only be checked every 10 seconds so queries will not
  # be aborted immediately when hitting the limit.
  max-select-point = 0

  # The maximum number of series a SELECT can run.  A value of 0 will make the maximum series
  # count unlimited.
  max-select-series = 0

  # The maximum number of group by time bucket a SELECT can create.  A value of zero will max the maximum
  # number of buckets unlimited.
  # max-select-buckets = 0

###
### [retention]
###
### Controls the enforcement of retention policies for evicting old data.
###

[retention]
  # Determines whether retention policy enforcement enabled.
  # enabled = true
```

```
  # The interval of time when retention policy enforcement checks run.
  # check-interval = "30m"
[shard-precreation]
  # Determines whether shard pre-creation service is enabled.
  # enabled = true

  # The interval of time when the check to pre-create new shards runs.
  # check-interval = "10m"

  # The default period ahead of the endtime of a shard group that its successor
  # group is created.
  # advance-period = "30m"


[monitor]
  # Whether to record statistics internally.
  # store-enabled = true

  # The destination database for recorded statistics
  # store-database = "_internal"

  # The interval at which to record statistics
  # store-interval = "10s"

[http]
  # Determines whether HTTP endpoint is enabled.
  # enabled = true

  # The bind address used by the HTTP service.
  # bind-address = ":8086"

  # Determines whether user authentication is enabled over HTTP/HTTPS.
  # auth-enabled = false

  # The default realm sent back when issuing a basic auth challenge.
  # realm = "InfluxDB"

  # Determines whether HTTP request logging is enabled.
  log-enabled = false

  # Determines whether detailed write logging is enabled.
  write-tracing = false

  # Determines whether the pprof endpoint is enabled.  This endpoint is used for
  # troubleshooting and monitoring.
  # pprof-enabled = true

  # Determines whether HTTPS is enabled.
  # https-enabled = false

  # The SSL certificate to use when HTTPS is enabled.
  # https-certificate = "/etc/ssl/influxdb.pem"

  # Use a separate private key location.
  # https-private-key = ""

  # The JWT auth shared secret to validate requests using JSON web tokens.
  # shared-secret = ""

  # The default chunk size for result sets that should be chunked.
  # max-row-limit = 0
```

```
  # The maximum number of HTTP connections that may be open at once.  New connections that
  # would exceed this limit are dropped.  Setting this value to 0 disables the limit.
  # max-connection-limit = 0

  # Enable http service over unix domain socket
  # unix-socket-enabled = false

  # The path of the unix domain socket.
  # bind-socket = "/var/run/influxdb.sock"
[subscriber]
  # Determines whether the subscriber service is enabled.
  # enabled = true

  # The default timeout for HTTP writes to subscribers.
  # http-timeout = "30s"

  # Allows insecure HTTPS connections to subscribers.  This is useful when testing with self-
  # signed certificates.
  # insecure-skip-verify = false

  # The path to the PEM encoded CA certs file. If the empty string, the default system certs will be used
  # ca-certs = ""

  # The number of writer goroutines processing the write channel.
  # write-concurrency = 40

  # The number of in-flight writes buffered in the write channel.
  # write-buffer-size = 1000
…..
[continuous_queries]
  # Determines whether the continuous query service is enabled.
  # enabled = true

  # Controls whether queries are logged when executed by the CQ service.
  # log-enabled = true

  # interval for how often continuous queries will be checked if they need to run
  # run-interval = "1s"
```